CS 3110
Data Structures and Functional Programming
Lecture 1 - Course Overview

Benjamin Ylvisaker
Cornell University
Spring 2013
Course Staff

• Instructor: Benjamin Ylvisaker

• “Visiting” from GrammaTech, Inc

• Into: software engineering, programming languages, concurrency and parallelism

• Wrote a few thousand lines of O’Caml in grad school

• TAs and Consultants: Many

• We have a large and talented team. Make use of them!

• Office hours: on the website
Course Meetings

- Lectures: Tuesday & Thursday 10:10-11:00am

- Sections: Monday & Wednesday
  - Meeting at the end of lecture today to schedule TBD section

- New material in lecture and section

- Attendance is expected
  - Good attendance buys you priority when there is high demand for staff attention
Course Website

- http://www.cs.cornell.edu/Courses/cs3110/2013sp/
- Course material
- Problem sets
- Announcements

- Course notes are fairly detailed, but course staff may say things in lecture and section that you will be expected to know
- Notes will be available before lecture/section
- If you find the course challenging, try to at least skim them in advance
- Notes are not a replacement for attendance
Piazza and CMS

- **Piazza**
  - Online discussion forums
  - Monitored by TAs/consultants
  - Ask for help, but do not post solutions

- **CMS**
  - “Course Management System”
  - Built by Andrew Myers et al.
  - Assignments and grades posted there
Coursework

- 6 Problem sets
  - Due Thursdays at 11:59pm
  - Short, random grace period
- PS 1 (out today) is due Thursday, Jan 31
- Electronic submission via CMS
- 4 individual, 2 with teams of 2
  - 3 weeks for big assignments, with checkpoints
- 2 preliminary exams and a final
Grading

• Breakdown:
  • 50% Problem sets
  • 30% Preliminary exams
  • 20% Final exam

• Final grades determined by a combination of your scores relative to the class distribution and the success of the class overall
Late Policy

- You may submit up to when we start grading
  - 15 point penalty (out of 100) per day
  - As soon as we start grading, zero
- Submit early and often
  - CMS is your friend
  - Code that fails to compile will likely get a zero
- When emergencies come up, talk to Ben ASAP
Academic Integrity

- Strictly enforced
- We check your work for similarity
- If you do not do your own work, it will be unpleasant and painful for everyone involved
- To avoid temptation, start early
  - Learning new programming ideas often requires sleeping on it
  - Office hours almost every day of the week
- Course staff is here to help
What this Course is About

• Programming isn’t hard

• Programming well is very hard
  • 10x range in effectiveness

• We want you to write code that is
  • Reliable, efficient, readable, testable, provable, maintainable, ... beautiful

• Expand your problem solving skills
  • Recognize problems and map them onto known good solutions
Thinking Versus Typing

• “A year at the lab bench saves an hour at the library”

• Fact: There are infinitely many wrong programs

• Corollary: If your program isn’t working and you don’t know why, making random tweaks is unlikely to help

• If you find yourself changing “<“ to “<=” in hopes that your code will work, you’re in trouble

• Lesson: Think before you type!
CS 3110 Challenges

• Some of you have gotten away with bad habits in previous programming classes
  • Just hack until it works
  • Solve everything by yourself
  • Write first, test later

• CS 3110 ≈ Tour de France
  • Professionals need good work habits and the right approach
  • We need to think rigorously about programs and their models
  • Think for a few minutes, instead of typing for a few days!
Rule #1

• Good programmers are lazy
• Never write the same code twice ("DRY")
• Reusable libraries
• Keep interfaces small and simple

• Pick a framework that makes it easy to write the code you need
• Early focus on speed is a disaster

• Rapid prototyping
Main Goal of CS 3110

• Master key linguistic abstractions:
  • Procedural abstraction
  • Control: iteration, recursion, pattern matching, laziness, exceptions, events
  • Encapsulation: closures, ADTs
  • Parameterization: higher-order procedures, modules

• Mostly in service to rule #1

• Transcends individual programming languages
Other Goals

• Exposure to software engineering techniques:
  • Modular design
  • Integrated testing
  • Code reviews

• Exposure to abstract models:
  • Models for design & communication
  • Models & techniques for proving correctness
  • Models for analyzing space & time

• Rigorous thinking about programs!
  • Proofs, like in high school geometry
Choice of Language

- This matters less than you suspect
- Must be able to learn new languages
  - This is relatively easy if you understand programming models and paradigms
- We will be using OCaml, a dialect of ML
- Why use yet another language?
  - Not to mention an obscure one?
- Main answer: OCaml programs are easy to reason about
Why OCaml? 

- Awesome OCaml feature: many common errors simply impossible
  - More precisely, they are caught at compile time
  - Early failure is very important (why?)
- Functional language
  - Programs have a clear semantics
  - Heavy use of recursion
  - Lots of higher-order functions
  - Few side effects
- Statically typed and type safe
  - Many bugs caught by compiler
Imperative (Procedural) Programming

- Program uses commands (a.k.a statements) that do things to the state of the system:
  - \( x = x + 1; \)
  - \( a[i] = 42; \)
  - \( p.next = p.next.next; \)
- Functions and methods can have side effects
  - \( \text{int wheels(Vehicle v)} \) { \( \text{v.size}++ \); \( \text{return v.numw;} \) }
Functional Style (1/2)

- Idea: program without side effects
- Effect of a function is only to return a result value
- Program is an expression that can be evaluated to produce a value
- For example, evaluating 2+2 yields 4
- Just like mathematical expressions
- Enables equational reasoning about programs:
  - if x equals y, replacing y with x has no effect:
  - let x=f(0) in x+x equivalent to f(0)+f(0)
Functional Style (2/2)

- Bind variables to values, don’t mutate existing variables
- No concept of $x=x+1$ or $x++$
- These do nothing remotely like $x++$
  - `let x = x+1` in `x`
  - `let rec x = x+1` in `x`
- The former assumes an existing binding for $x$ and creates a new one (no modification of $x$)
- The latter is an invalid expression
Trends in Industry Encouraging Functional Style

• Fantasy: program interacts with a single system state
  • Interactions are reads from and writes to variables or fields.
  • Reads and writes are very fast
  • Side effects are instantly seen by all parts of a program

• Reality: there is no single state
  • Multicores have own caches with inconsistent copies of state
  • Programs are spread across different cores and computers (PS5 & PS6)
  • Side effects in one thread may not be immediately visible in another
  • Imperative languages are a bad match to modern hardware
Imperative vs Functional

• Functional programming languages strongly encourage
  • Building code out of small functions
  • f(x) always gives the same result for the same x
  • Node side effects: easier to reason about programs
  • Better fit for modern hardware

• Functional style usable in Java, C, Python, ...
  • Example: “Lambda” support in C++
  • Often harder to stick with a purely functional style
Programming Language Map

- **Functional**
  - Lisp
  - Scheme
  - Haskell
  - OCaml
  - SML

- **Imperative**
  - Fortran
  - Matlab
  - C
  - C++
  - Java
  - JavaScript
  - Perl
  - Pascal

- **Object-Oriented**
  - ML family

- **ML family**
OCaml Example 1

• let rec sumsq n =
  if n=0 then 0
  else n*n + sumsq(n-1)
OCaml Example 2

- let rec sumop f n =
  if n=0 then 0
  else f n + sumop f (n-1)
- sumop cube 5
- sumop (function x -> x*x*x) 5
OCaml Example 3

- let rec reverse lst =

  match lst with
  | [] -> []
  | h::t -> reverse t @[h]
OCaml Example 4

- let rec reverse lst =

  match lst with
  | [] -> []
  | h::t -> reverse t @[h]
Why OCaml?

- Objective Caml is one of the most robust and general functional languages available
  - Used in financial industry
  - Lightweight and good for rapid prototyping
- Embodies important ideas better than Java, C++
  - Many of these ideas work in Java, C++, and you should use them...
- Learning a different language paradigms will make you a more flexible programmer down the road
  - Likely that Java and C++ will be replaced
  - Principles and concepts beat syntax
  - Ideas in ML will likely be in next-generation languages
Rough Schedule

- Introduction to functional programming (6)
- Functional data structures (5)
- Verification and Testing (5)
- Preliminary Exam #1
- Concurrency (1)
- Data structures and analysis of algorithms (5)
- Preliminary Exam #2
- Topics: streams, λ-calculus, garbage collection
- Final exam